# Contents

## 1 Question 1

### 1.1 Briefly describe the requirements of a 3rd normal form (3NF)

1. A table must not contain any repeating columns
2. Every non-key column must depend only on the primary key

### 1.2 Briefly describe M:N relationship between two entities (share an ER model)

A film can belong to multiple categories (e.g., action, drama, comedy), and a film category can have multiple films associated with it.

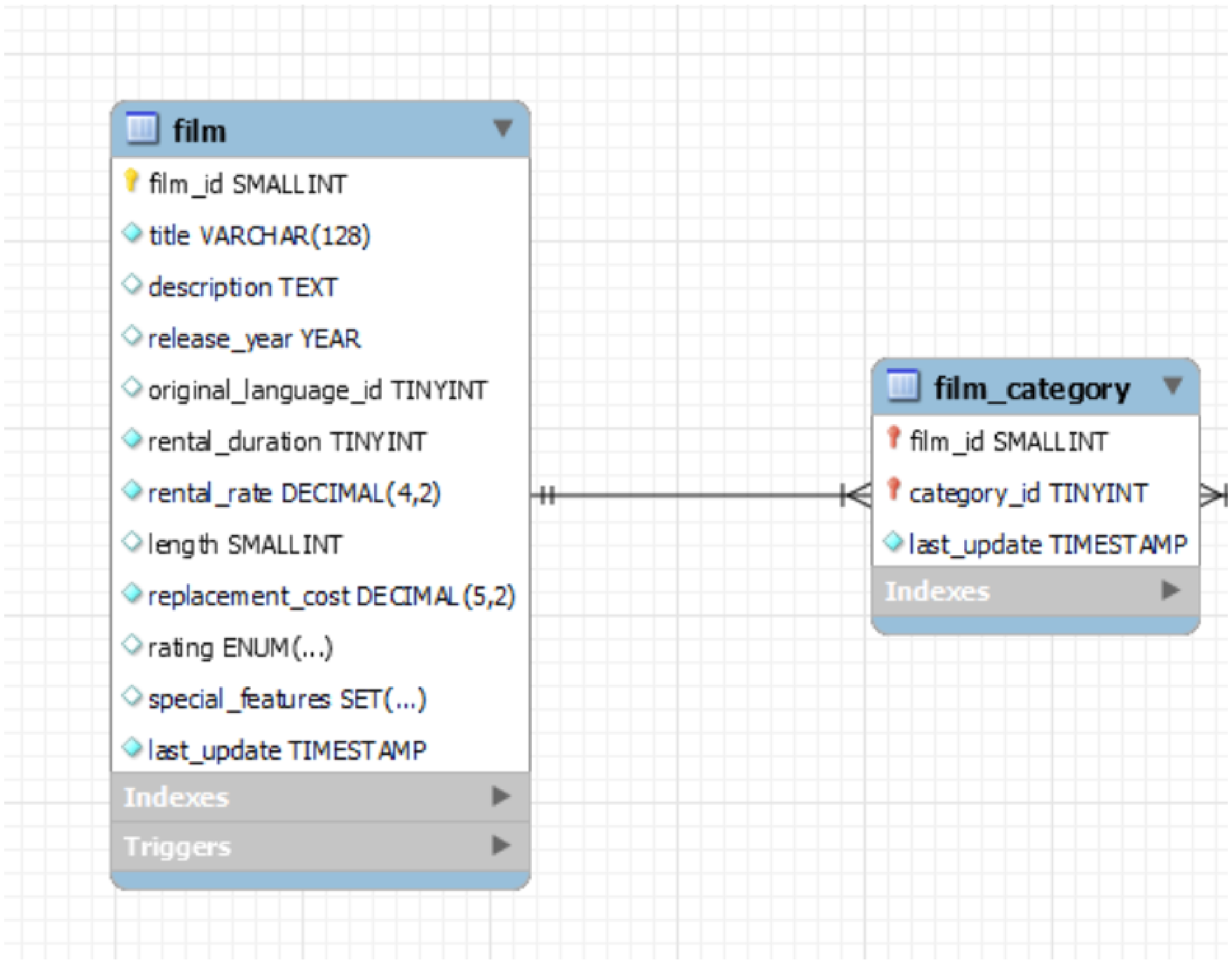


Figure 1: M:N ER-model (film and film category)

### 1.3 Briefly describe when “WHERE” clause can and cannot be used to select rows after a “GROUP BY” clause?

1. ”WHERE” clause can used to filter rows before they are grouped by the ”GROUP BY” clause.

2. cannot use "WHERE" to filter aggregated results or groups after "GROUP BY". For that, we use the "HAVING" clause

### 1.4 Briefly discuss three differences between normalized modeling (OLTP database) and dimensional modeling (OLAP database)

- Data Content and Structure
  - OLTP: Contains current values and is structured for transactional efficiency, usually using Entity-Relationship (ER) models.
  - OLAP: Contains historical, derived, and summarized data and is structured for complex query efficiency, typically using Star or Snowflake schemas.
- Data Volume and Access Type
  - OLTP: Typically holds MB/GB of data and supports frequent read, update, and delete operations.
  - OLAP: Holds much larger volumes (GB/TB/PB) of data and is primarily read-only.
- Data Usage and Response Time
  - OLTP: Has predictable and repetitive usage patterns, with sub-second response times expected.
  - OLAP: Supports ad hoc, random, heuristic queries, which may take several seconds to minutes to

### 1.5 Briefly discuss two differences between the NoSQL document database and a NoSQL wide column database.

- Data Organization
  - Document-based: Data is stored in documents, typically in formats like JSON. Each document is a semi-structured set of key-value pairs.
  - Column-based: Data is organized by columns and partitioned into column families. Each column family stores its data in separate files.
- Access and Indexing:
  - Document-based: Documents are primarily accessed via their unique IDs, but secondary indexes can be used for rapid access using other attributes.
  - Column-based: These databases emphasize columnar access and can version data values, allowing for historical data retrieval.

### 1.6 Briefly discuss what the following command returns when used on documentDB: db.restaurants.find("address.location": location)

Will search the "restaurants" collection and return all restaurants where the embedded "location" field inside the "address" object matches the value specified by the location variable.

## 1.7 Briefly discuss the difference between “scan” and “get” in HBase

- Functionality:
  - scan: Fetches all data in a table.
  - get: Fetches a specific item based on its rowid.
- Scope:
  - scan: Retrieves all rows in the table.
  - get: Retrieves a single row identified by the given rowid.
- Performance:
  - scan: Potentially slower as it reads the entire table.
  - get: Faster as it targets a specific row.

## 1.8 Briefly discuss why NoSQL databases are not good for “join” and “group” operations?

1. Design Philosophy: NoSQL databases are designed for horizontal scalability. Distributed joins across multiple nodes are complex and inefficient.
2. Schema Flexibility: NoSQL databases, especially document-based ones, allow for heterogeneous schemas. This flexibility complicates join operations which require consistent schemas.
3. Performance: Joining data often requires merging datasets, which is resource-intensive. NoSQL databases prioritize quick, direct access over complex operations.
4. Denormalization: Many NoSQL databases encourage denormalization, storing related data together instead of referencing it, reducing the need for joins.

## 1.9 Briefly discuss why NoSQL databases are not good for “join” and “group” operations?

1. Traffic and Volume: TikTok handles vast amounts of data due to its user-generated content. This may demand larger region sizes to prevent too frequent region splits which can impact performance.
2. Parallel Scalability: Setting the max size too low can cause excessive region splits, leading to many small regions. This might hinder parallel processing capabilities across the nodes.
3. Performance: A too high max size might delay region splits, causing certain regions to handle excessive loads, leading to slower performance.

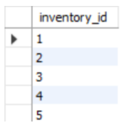
Considering these factors, while the default size is 256 MB, TikTok might benefit from a slightly larger region size to reduce the frequency of splits while ensuring that the size doesn’t get so large that it hinders performance. An ideal size might be in the range of 512 MB to 1 GB, but it’s essential to continuously monitor and adjust based on real-world performance metrics.

## 2 Question 2

### 2.1 Check if a movie is in stock: list inventory ids that are currently in stock across all locations for movie ACADEMY DINOSAUR

Listing 1: 2.1

```
-- question 2.1
with out_stock as(
select distinct inventory_id
from rental
where return_date is null)

select inventory_id from film a
left join inventory b
on a.film_id = b.film_id
where a.title like 'ACADEMY␣DINOSAUR'
and b.inventory_id not in (select inventory_id from out_stock)
```



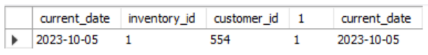
```
inventory_id
1
2
3
4
5
```

Figure 2: 'ACADEMY DINOSAUR'

## 2.2 For movie in stock, in the customer location, checkout the movie (in rental table) for customer = “DWAYNE OLVERA” (assume your staff id = 1).

Listing 2: 2.2

```sql
insert into rental (rental_date, inventory_id, customer_id, staff_id, last_update)
select current_date , i.inventory_id, c.customer_id, 1, current_date
from inventory i
join film f on i.film_id = f.film_id
join customer c on c.first_name = 'DWAYNE' and c.last_name = 'OLVERA'
where f.title = 'ACADEMY␣DINOSAUR'
and i.inventory_id not in (
    select r.inventory_id
    from rental r
    where r.return_date is null
)
limit 1;
```



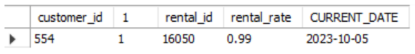
| current_date | inventory_id | customer_id | 1 | current_date |
|---|---|---|---|---|
| 2023-10-05 | 1 | 554 | 1 | 2023-10-05 |

Figure 3: checkout the movie (in rental table) for customer

## 2.3 For rental entered, collect the rental payment for the movie and add a row in the payment table

Listing 3: 2.3

```
INSERT INTO payment (customer_id, staff_id, rental_id, amount, payment_date)
SELECT c.customer_id, 1, r.rental_id, f.rental_rate, CURRENT_DATE
FROM rental r
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
JOIN customer c ON r.customer_id = c.customer_id
WHERE c.first_name = 'DWAYNE' AND c.last_name = 'OLVERA'
AND f.title = 'ACADEMY␣DINOSAUR'
AND r.last_update = CURRENT_DATE; -- This is to ensure we're capturing today's rental
```



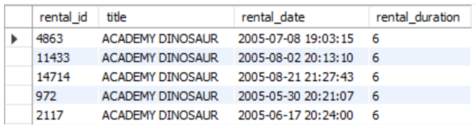
| customer_id | 1 | rental_id | rental_rate | CURRENT_DATE |
|---|---|---|---|---|
| 554 | 1 | 16050 | 0.99 | 2023-10-05 |

Figure 4: add a row in the payment table

## 2.4 Create a list of overdue movies (i.e., rental duration less than 2006-02-18– rental date)

Listing 4: 2.4

```sql
SELECT r.rental_id, f.title, r.rental_date, f.rental_duration
FROM rental r
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
WHERE DATE_ADD(r.rental_date, INTERVAL f.rental_duration DAY) < '2006-02-18';
```



| rental_id | title | rental_date | rental_duration |
|---|---|---|---|
| 4863 | ACADEMY DINOSAUR | 2005-07-08 19:03:15 | 6 |
| 11433 | ACADEMY DINOSAUR | 2005-08-02 20:13:10 | 6 |
| 14714 | ACADEMY DINOSAUR | 2005-08-21 21:27:43 | 6 |
| 972 | ACADEMY DINOSAUR | 2005-05-30 20:21:07 | 6 |
| 2117 | ACADEMY DINOSAUR | 2005-06-17 20:24:00 | 6 |

Figure 5: Create a list of overdue movies

## 3 Question 3

### 3.1 Load the positive and negative words data1,2 using "LOAD DATA LOCAL INFILE" into two separate tables: pos words and neg words.

Listing 5: load the pos words and neg words

```sql
-- 3.1
CREATE TABLE pos_words (
    word VARCHAR(255) PRIMARY KEY
);

LOAD DATA LOCAL INFILE '/Users/lanston/Downloads/pos_words.txt'
INTO TABLE pos_words
LINES TERMINATED BY '\n'
IGNORE 35 LINES;

CREATE TABLE neg_words (
    word VARCHAR(255) PRIMARY KEY
);

LOAD DATA LOCAL INFILE '/Users/lanston/Downloads/neg_words.txt'
INTO TABLE neg_words
LINES TERMINATED BY '\n'
IGNORE 35 LINES;
```

## 3.2 Write mySQL user-defined-functions that takes a text input and returns the number of positive and negative words

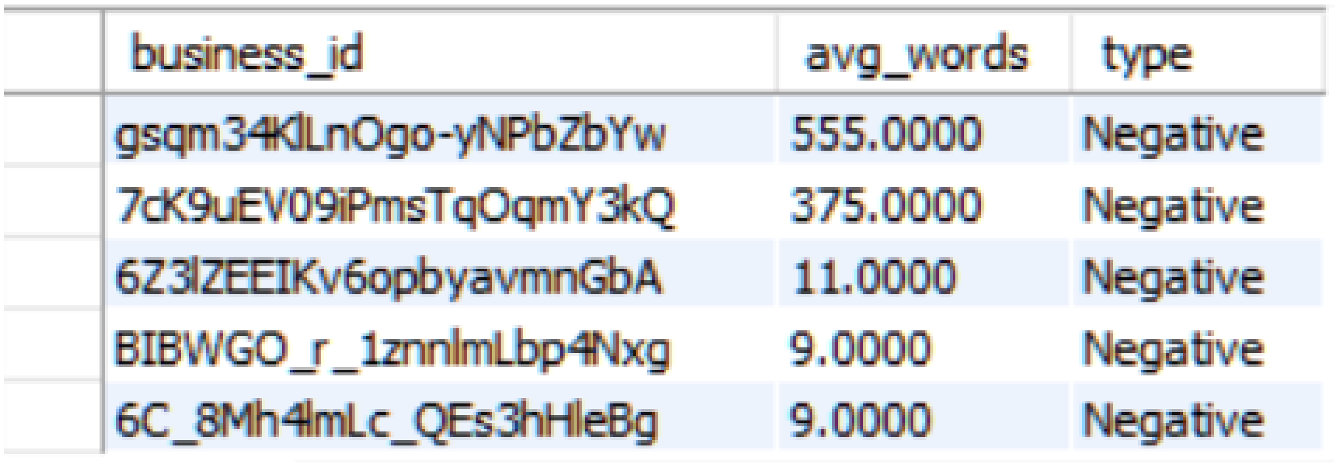
Listing 6: creaet user define function

```
DELIMITER //
CREATE FUNCTION CountPositiveWords(input_text TEXT)
RETURNS INT
BEGIN
  DECLARE pos_count INT DEFAULT 0;
  DECLARE word TEXT;
  DECLARE cleaned_text TEXT;

  -- Convert to lowercase and clean up the text: keep only characters a-z and A-Z, then
      normalize spaces
  SET input_text = LOWER(input_text);
  SET cleaned_text = TRIM(REGEXP_REPLACE(REGEXP_REPLACE(input_text, ’[^a-zA-Z␣]’, ’␣’),
      ’\\s+’, ’␣’));

  -- Process each distinct word in the cleaned text
  WHILE LENGTH(cleaned_text) > 0 DO
    IF LOCATE(’␣’, cleaned_text) > 0 THEN
      SET word = SUBSTRING(cleaned_text, 1, LOCATE(’␣’, cleaned_text) - 1);
      SET cleaned_text = REPLACE(cleaned_text, word, ’’); -- remove all occurrences of the word
    ELSE
      SET word = cleaned_text;
      SET cleaned_text = ’’;
    END IF;

    -- Check if the word exists in pos_words
    IF EXISTS (SELECT 1 FROM yelp_lanston.pos_words WHERE LOWER(word) = TRIM(word)) THEN
      SET pos_count = pos_count + 1;
    END IF;

  END WHILE;

  RETURN pos_count;
END //
DELIMITER ;


DELIMITER //
CREATE FUNCTION CountNegativeWords(input_text TEXT)
RETURNS INT
BEGIN
  DECLARE neg_count INT DEFAULT 0;
  DECLARE word TEXT;
  DECLARE cleaned_text TEXT;

  -- Convert to lowercase and clean up the text: keep only characters a-z and A-Z, then
      normalize spaces
  SET input_text = LOWER(input_text);
  SET cleaned_text = TRIM(REGEXP_REPLACE(REGEXP_REPLACE(input_text, '[^a-zA-Z␣]', '␣'),
      '\\s+', '␣'));

  -- Process each distinct word in the cleaned text
  WHILE LENGTH(cleaned_text) > 0 DO
    IF LOCATE('␣', cleaned_text) > 0 THEN
      SET word = SUBSTRING(cleaned_text, 1, LOCATE('␣', cleaned_text) - 1);
      SET cleaned_text = REPLACE(cleaned_text, word, ''); -- remove all occurrences of the word
    ELSE
      SET word = cleaned_text;
      SET cleaned_text = '';
    END IF;

    -- Check if the word exists in neg_words
    IF EXISTS (SELECT 1 FROM yelp_lanston.neg_words WHERE LOWER(word) = TRIM(word)) THEN
      SET neg_count = neg_count + 1;
    END IF;

  END WHILE;

  RETURN neg_count;
END //
DELIMITER ;
```

### 3.3 write a SQL query to show top 5 businesses with highest average positive words and highest average negative words

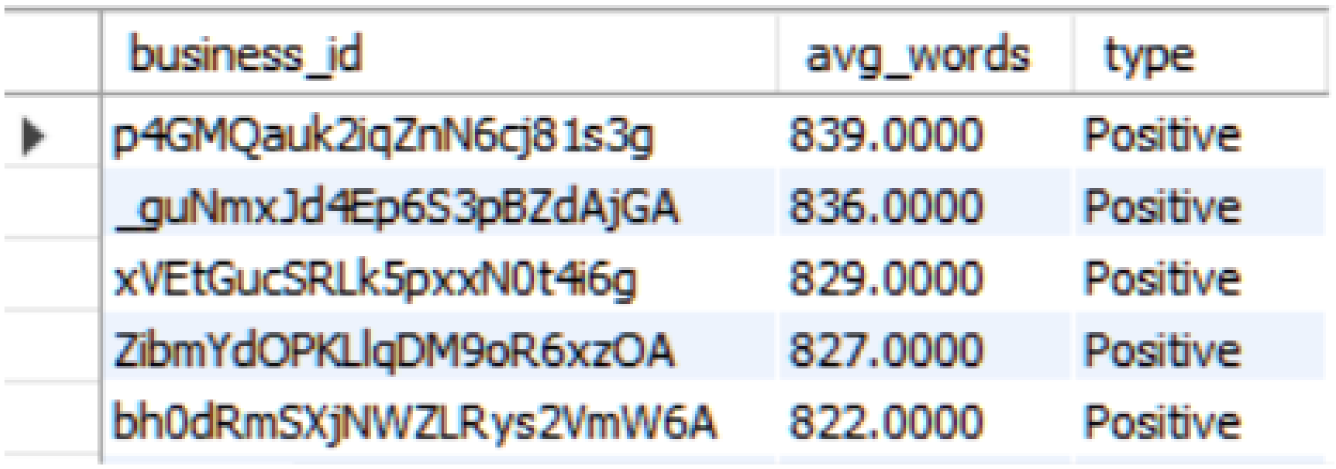
Listing 7: top 5 businesses

```sql
-- Top 5 businesses with the highest average positive words
SELECT business_id,
      AVG(CountPositiveWords(text)) AS avg_positive_count
FROM yelp.yelp_review
GROUP BY business_id
ORDER BY avg_positive_count DESC
LIMIT 5
UNION
-- Top 5 businesses with the highest average negative words
SELECT business_id,
      AVG(CountNegativeWords(text)) AS avg_negative_count
FROM yelp.yelp_review
GROUP BY business_id
ORDER BY avg_negative_count DESC
LIMIT 5;
```



| | business_id | avg_words | type |
|---|---|---|---|
| | gsqm34KlLnOgo-yNPbZbYw | 555.0000 | Negative |
| | 7cK9uEV09iPmsTqOqmY3kQ | 375.0000 | Negative |
| | 6Z3lZEEIKv6opbyavmnGbA | 11.0000 | Negative |
| | BIBWGO_r_1znnlmLbp4Nxg | 9.0000 | Negative |
| | 6C_8Mh4mLc_QEs3hHleBg | 9.0000 | Negative |

Figure 6: Top 5 Negative



| business_id | avg_words | type |
|---|---|---|
| p4GMQauk2iqZnN6cj81s3g | 839.0000 | Positive |
| _guNmxJd4Ep6S3pBZdAjGA | 836.0000 | Positive |
| xVEtGucSRLk5pxxN0t4i6g | 829.0000 | Positive |
| ZibmYdOPKLlqDM9oR6xzOA | 827.0000 | Positive |
| bh0dRmSXjNWZLRys2VmW6A | 822.0000 | Positive |

Figure 7: Top 5 Positive

### 3.4 Using the result of previous query as a subquery, write a query to show the sentiment of each review

Listing 8: 3.4

```
SELECT business_id, text,
       (CountPositiveWords(text) - CountNegativeWords(text)) /
       (LENGTH(text) - LENGTH(REPLACE(text, '␣', '')) + 1) AS sentiment
FROM yelp.yelp_review
WHERE business_id IN (
    -- Subquery from the previous section to get businesses
    SELECT business_id FROM (
        SELECT business_id,
               AVG(CountPositiveWords(text)) AS avg_positive_count
        FROM yelp.yelp_review
        GROUP BY business_id
        ORDER BY avg_positive_count DESC
        LIMIT 5
        UNION
        SELECT business_id,
               AVG(CountNegativeWords(text)) AS avg_negative_count
        FROM yelp.yelp_review
        GROUP BY business_id
        ORDER BY avg_negative_count DESC
        LIMIT 5
    ) AS SubQuery
)
ORDER BY business_id, sentiment DESC;
```

## 4 Question 4

### 4.1 Random row sample

Listing 9: 4.1

```
SELECT *
FROM big_table
WHERE RAND() < (SELECT 5 / COUNT(*) FROM big_table)
LIMIT 5;
```

### 4.2 Employee Salaries

Listing 10: 4.1

```
SELECT e.first_name, e.last_name, e.salary
FROM employees e
WHERE e.id IN (
    SELECT MAX(id)
    FROM employees
    GROUP BY first_name, last_name
);
```

### 4.3 Monthly Customer

Listing 11: 4.3

```
SELECT
    DATE_FORMAT(t.created_at, '%Y-%m-01') AS month,
    COUNT(DISTINCT t.user_id) AS num_customers,
    COUNT(DISTINCT t.id) AS num_orders,
    SUM(p.price * t.quantity) AS order_amt
FROM transactions t
JOIN products p ON t.product_id = p.id
GROUP BY month
ORDER BY month;
```

### 4.4 Closest SAT

Listing 12: 4.4

```
SELECT
    s1.student AS one_student,
    s2.student AS other_student,
    ABS(s1.score - s2.score) AS score_diff
FROM scores s1
JOIN scores s2 ON s1.id < s2.id
```

```
ORDER BY score_diff ASC, s1.id ASC, s2.id ASC
LIMIT 1;
```

## 4.5 Product Recommendation

Listing 13: 4.5

```
SELECT
    (SELECT name FROM products WHERE id = t1.product_id) AS P1,
    (SELECT name FROM products WHERE id = t2.product_id) AS P2,
    COUNT(*) AS count
FROM transactions t1
JOIN transactions t2 ON t1.user_id = t2.user_id AND t1.product_id < t2.product_id
GROUP BY t1.product_id, t2.product_id
ORDER BY count DESC
LIMIT 100;
```

# 5 Question 5

## 5.1 Database Design (ER Model)

- SportsTeams:
  - teamID (PK) ;teamName ;description ;headCoach ;yearOfEstablishment
- Facilities:
  - facilityID (PK) ;facilityName ;capacity ;constructionDate ;location ;lastInspectionDate
- Games:
  - gameID (PK) ;homeTeamID (FK to SportsTeams) ;opponentName ;attendance ;gameDate ;homeScore ;opponentScore ;homeGameFlag ;facilityID (FK to Facilities) ;gameType
- Customers:
  - customerID (PK) ;firstName ;lastName ;streetAddress ;city ;state ;zip ;email ;telephone
- SeasonTickets:
  - seasonTicketID (PK) ;customerID (FK to Customers) ;teamID (FK to SportsTeams) ;purchaseDate ;ticketPrice ;quantity

Assumptions:

1. Each game is between Emory and another external team, which is why the opponent is not another team from the same table.
2. Each season ticket purchase is for a specific sport, and the same customer can have multiple entries if they buy season tickets for different sports.
3. Game ticket prices can vary, hence price is recorded in the GameTickets table.The game date determines the season, year, month, etc.

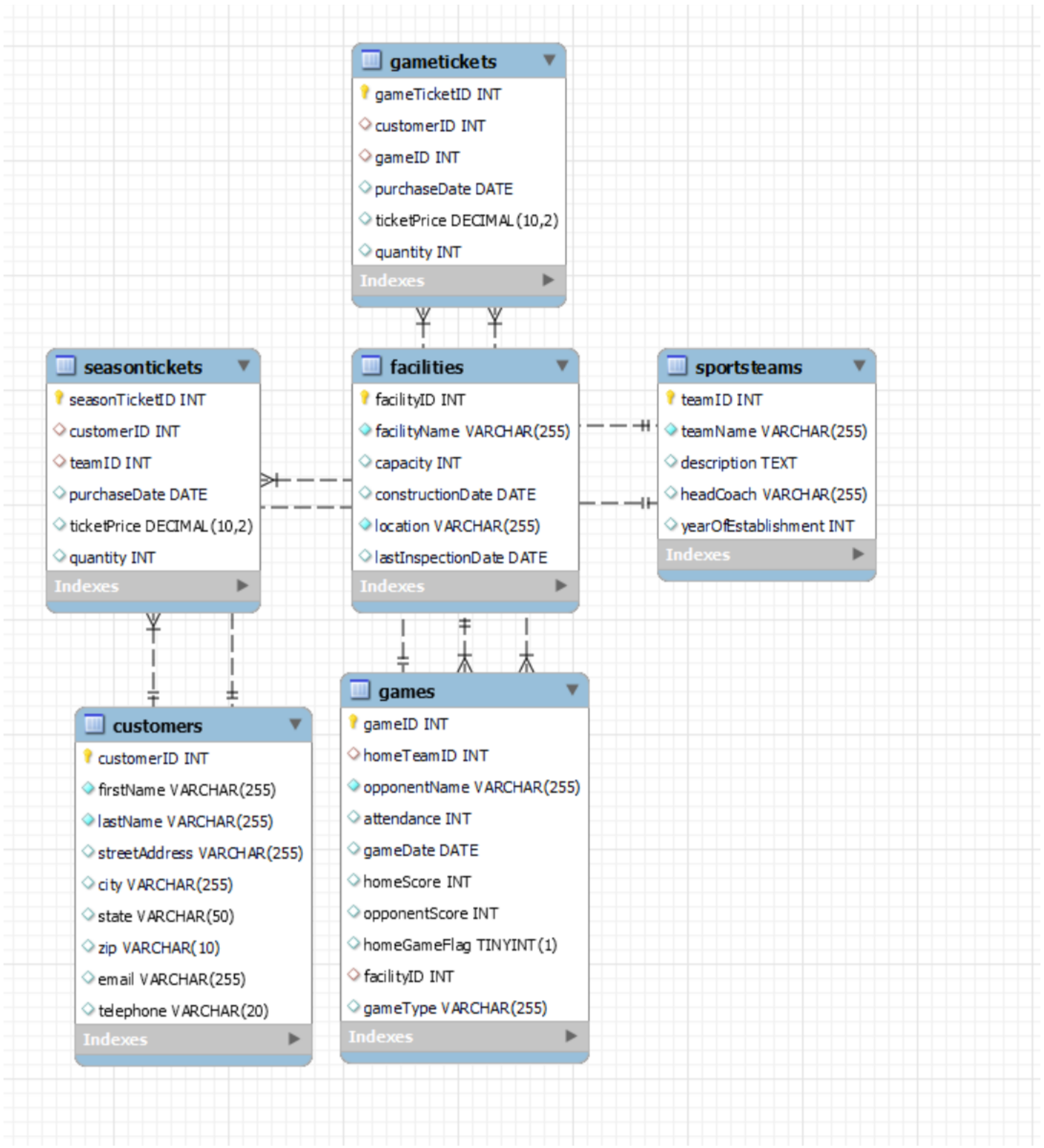


Figure 8: ER-model

## 5.2 Dimensional Modeling

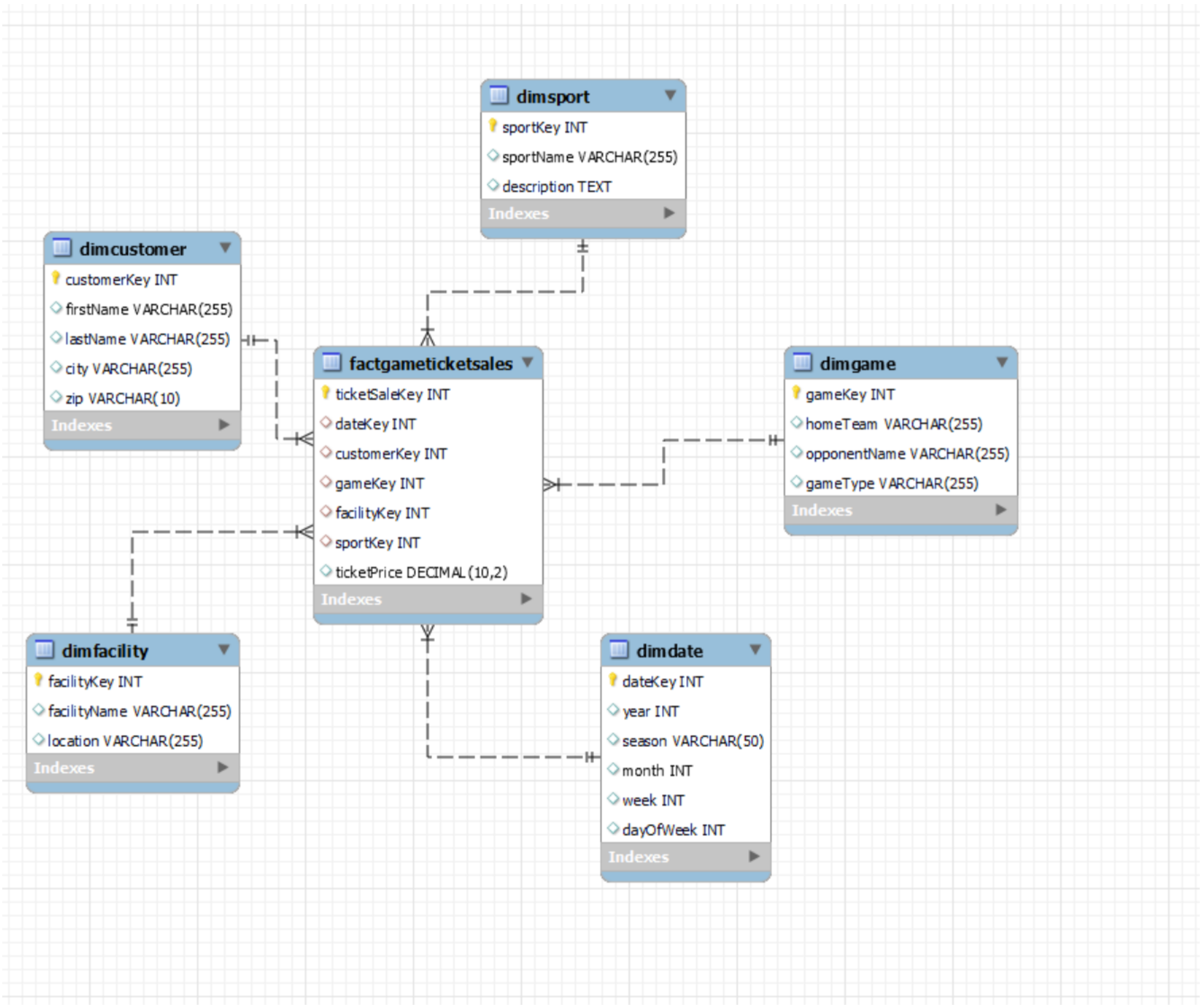


Figure 9: Dim table of ER-model

Fact Table (Grain: Each row represents a single game ticket sale):

- GameTicketSales:
  - ticketSaleID (PK) ;dateKey (FK) ;customerKey (FK) ;gameKey (FK) ;facilityKey (FK) ;sportKey (FK) ;ticketPrice
- Date:
  - dateKey (PK), year, season, month, week, dayOfWeek.
- Customer:
  - customerKey (PK), firstName, lastName, city, zip.
- Facility:

  - facilityKey (PK), facilityName, location.

- Sport:

  - sportKey (PK), sportName, description.

In my star schema, a row in the FactGameTicketSales table represents a single ticket sale for a specific game on a specific date to a specific customer, capturing details about the facility where the game was played and the sport type. This means for every individual ticket sale, there's a unique entry in my fact table, capturing its various dimensions and the corresponding sale amount (ticketPrice).

# 6 NoSQL Databases

Given the nature of the data to be stored, there are good arguments for using both wide-column and document databases. Here's a detailed examination of the scenario:

## 6.1 Database Design for Bing Chat:

In the context of the Bing chat, we are considering data that consists of user queries, system responses (which could be text or images), timestamps, and links clicked from the Bing search engine. This sort of data is semi-structured and can be variable in nature.

**Document-based NoSQL database (like MongoDB):** It can store structured data in JSON-like format. Each chat interaction can be represented as a single document with fields for the user question, system response, timestamp, and clicked links.

**Wide-column store (like Cassandra):** Given that Bing search data is already stored in this format, it may make sense from an integration or consistency perspective to use the same for chat logs. Each user interaction can be a new row, with columns for each piece of data (user question, response, timestamp, clicked links, etc.). Given the flexibility of wide-column stores, columns can be added as required without a rigid schema.

**Attributes:**

- **ChatID (Primary Key):** Unique identifier for each chat.
- **UserID:** Unique identifier for the user.
- **Question:** Text of the user's question.
- **Response:** Text or image link of the system's response.
- **Timestamp:** DateTime when the interaction occurred.
- **ClickedLinks:** Array or set of URLs clicked from the Bing search engine.

## 6.2 CAP Properties:

The two most essential CAP properties for this use case would be:

- **Consistency:** Ensure that all replicas have the same data, given the importance of user interactions and responses.
- **Availability:** Ensure that the system remains operational and responsive to user requests, considering the customer-facing nature of the application.

## 6.3 LinkedIn Integration with OpenAI:

Since LinkedIn already uses Voldemort (a key-value store), for the sake of integration and simplicity, one might opt to continue using the same key-value structure.

**Key:** Unique InteractionID.

**Value:**

- **UserID:** Unique identifier for each LinkedIn user.
- **API_Call:** Specific OpenAI API endpoint that was called.
- **Timestamp:** DateTime when the API call was made.
- **Response:** Result of the API call, either generated text or image link.

## 6.4 Data Warehouse Design (Star Schema):

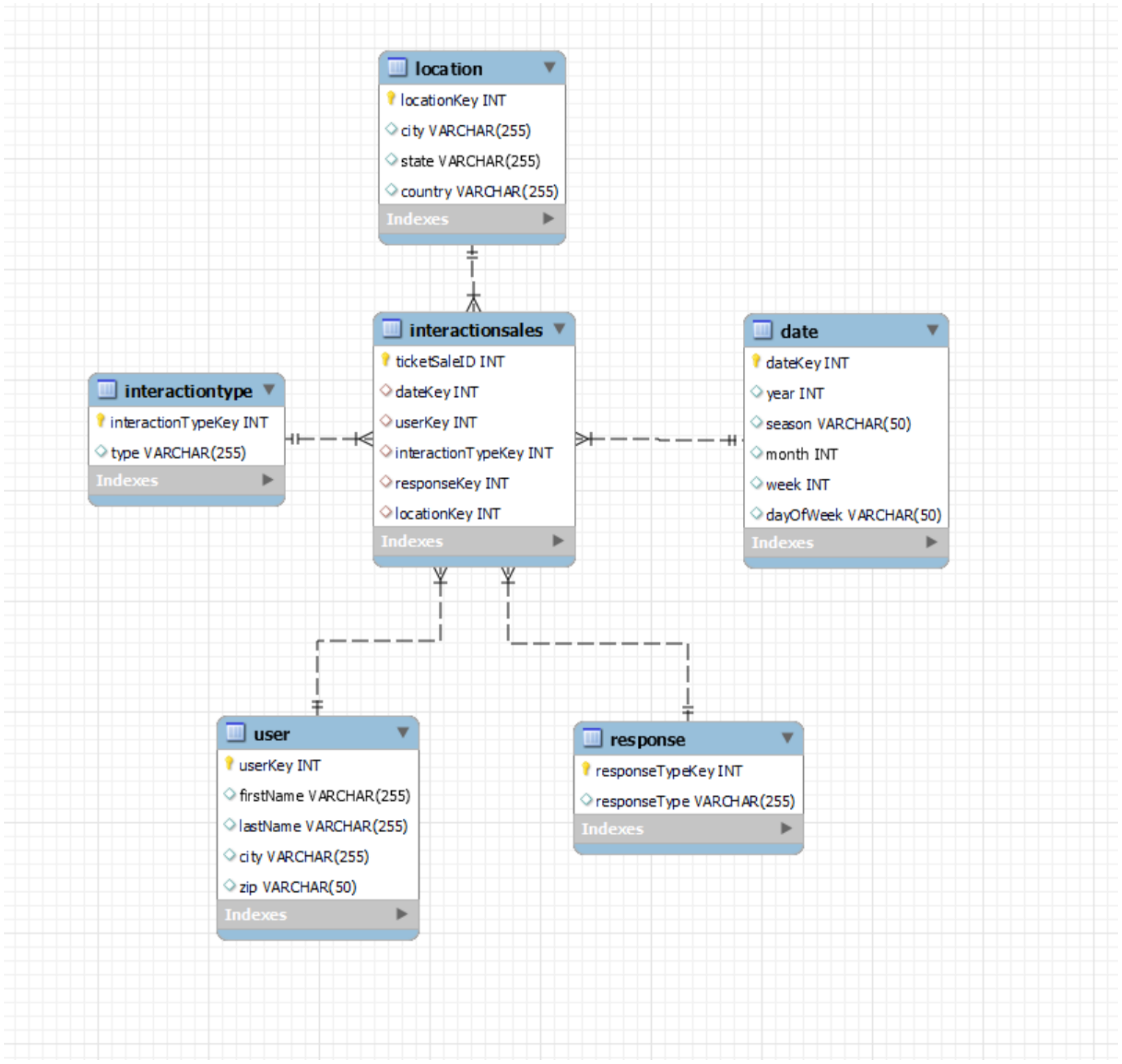


Figure 10: ER-model of Star Schema

**Fact Table:**

- **InteractionSales:** ticketSaleID, dateKey, userKey, interactionTypeKey, responseKey, locationKey.

**Dimension Tables:**

- **Date:** dateKey, year, season, month, week, dayOfWeek.
- **User:** userKey, firstName, lastName, city, zip.
- **InteractionType:** interactionTypeKey, type (Bing Chat, LinkedIn).
- **Response:** responseTypeKey, responseType (text, image, etc.).

- **Location:** locationKey, city, state, country.

**ETL Process:**

1. Extract data from both the Bing's document-based and/or wide-column store and LinkedIn's Voldemort key-value store.
2. Transform this data to fit the star schema.
3. Load the transformed data into the data warehouse's fact and dimension tables.

**Why not use both wide-column and document databases?**

The primary advantage of using both types simultaneously is to leverage the strengths of each. While wide-column stores like Cassandra can handle massive amounts of data and are optimized for queries over